

PROGRAMMING
EXPERT

FAQ

Q What are 'strong' and 'weak' typing?

A Programming languages let you store data in named containers called variables. `Total = 49` stores the number 49 in a variable named 'Total'. Languages with weak typing can store different types of data in the same variable. So, in JavaScript, you can write:

```
Total = 49;
Total = "Shopper";
```

even though the values are different types of data (an integer and some text). With strong typing you must declare the type of data you'll store in a variable. For example, in C#, to store the integer 49 you'd use:

```
int Total;
Total = 49;
```

This helps the computer to spot bugs in your code.

Mike James

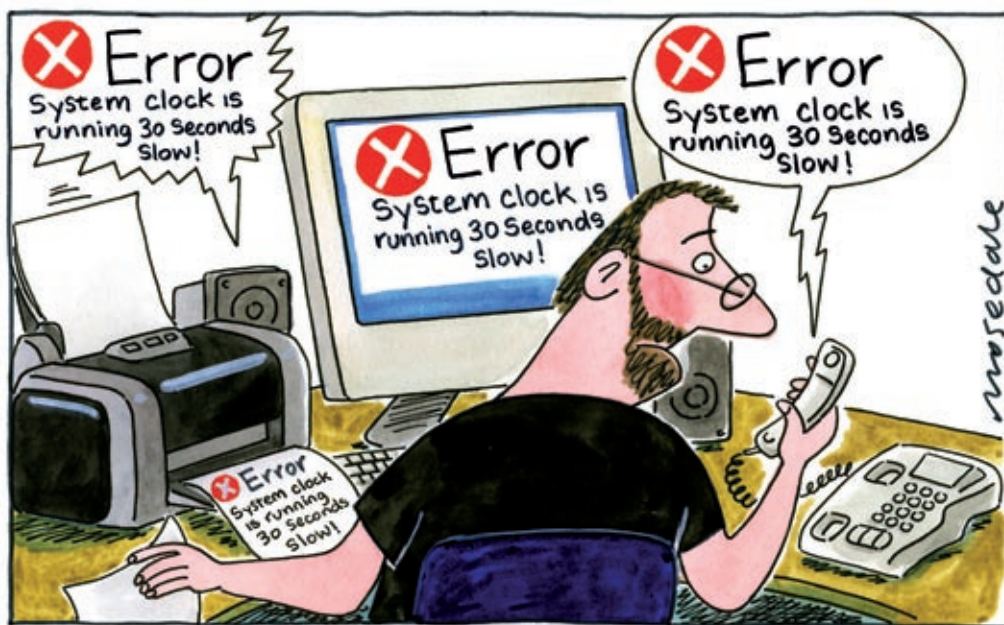
IT author, lecturer
and programmer

mike@computershopper.co.uk



At your service

Services are the household staff of Windows, working quietly in the background without bothering the user. Mike James sets them to work with a program that keeps an eye on his logs



Most everyday Windows programs interact with users, but not all do. A Windows service is a program that performs a background task with no user involvement. Services can be set to start automatically when a PC is turned on, before a user has even logged on. There are lots of standard services that provide facilities such as networking that are at the very heart of the Windows operating system, but it's fairly easy to create your own.

Why would you want to? Services are ideal when you want an application to run no matter which user is logged on, and to provide a facility that is available from the moment a machine is powered up. As an example, we'll create a simple service that reads the machine's event log and sends an email if it finds any error messages. We'll see how easy the .NET framework makes services, and how simple event logging is, too.

A SIMPLE SERVICE

The simplest way to create a service is to use Visual Studio's service template. This isn't included in the free Express Edition or the Standard Edition, though, so many programmers wrongly conclude that they can't use these versions to create services. In fact, creating a service without the template is fairly easy – so much so that it's a mystery why Microsoft reserved the template for the more expensive editions of Visual Studio.

There are two key considerations in the creation of a service. First you must start with the supplied `ServiceBase` class, which provides all the methods and properties you need to implement a service. Creating your service is just a matter of deriving a new class from `ServiceBase`, then customising it to make it do exactly what you want. Second, you must use a supplied installation class along with the .NET tool `InstallUtil` to install your service. You can find `InstallUtil` in the `Windows\Microsoft.NET\Framework\Vx.y.zzzz` directory (where `Vx.y.zzzz` is the latest version of the framework). When you type the command:

```
InstallUtil MyProg.exe
```

`IntallUtil` looks for any classes in `MyProg.exe` that carry the `[RunInstaller(true)]` attribute, and if it finds any, it runs the class constructor. This is a fairly general mechanism for installing applications, but we will use it to install our service.

To start, create a new application using C# Express – if you'd rather use VB or any other .NET language the changes you need to make are small. The easiest thing to do is use the Empty Project template and call the project `SimpleService`. Next, add a `CodeFile` to the project called `SimpleServer`. Creating the service is very easy but we need to use some unusual parts of the .NET Framework class library. Add a reference (Project>Add Reference) to 'System' and 'System.ServiceProcess', then add to the start of the program:

```
using System;
using System.ServiceProcess;
```

As I've already explained, the service will be implemented as a class that inherits from `ServiceBase`. Its constructor is called when the service is created and so consists of a few statements that set some fairly obvious standard properties:

```
public class SimpleServer : ServiceBase
{
    public SimpleServer()
    {
        this.ServiceName = "SimpleServer";
        this.CanStop = true;
        this.CanPauseAndContinue = false;
        this.AutoLog = true;
    }
}
```

The real work of the service is done by the `OnStart` and `OnStop` event handlers, which are called, as you might

expect, when the service is started and stopped using the usual Windows Services Administrative Tool. Exactly how this fits together will become clear as we progress.

```
protected override void
    OnStart(string[] args)
{
    // do startup stuff
}

protected override void OnStop()
{
    // do shutdown stuff
}
```

The only other method the class needs is Main, which is the default startup point of any console program. In this case Main simply creates an instance of the service class:

```
public static void Main()
{
    ServiceBase.Run(new
        SimpleServer());
}
```

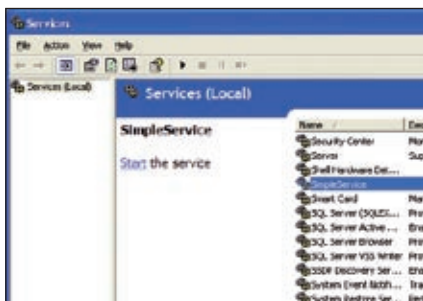
That's all there is to a Windows service, but we also need an installer to set up the Registry entries needed to make the system aware of your new service. You could do this manually or write a custom script for it, but the .NET framework provides a standard way of doing the job. First you must add a class that inherits from the .NET Installer class to your project. This then has to be marked with the RunInstaller attribute to make it instantiate during installation:

```
[RunInstaller(true)]
public class SimpleService :
    Installer
```

The only method you need to implement is the constructor, and this simply sets some easy-to-understand properties and creates two installer objects:

```
private ServiceProcessInstaller
    processInstaller;
private ServiceInstaller
    serviceInstaller;
public SimpleService()
{
    processInstaller = new
        ServiceProcessInstaller();
    serviceInstaller = new
        ServiceInstaller();

    processInstaller.Account =
        ServiceAccount.LocalSystem;
```



▲ Use the SimpleService tool from the Control Panel to start, stop and modify your service

```
serviceInstaller.StartType =
    ServiceStartMode.Manual;
serviceInstaller.ServiceName =
    "SimpleService";

Installers.Add(serviceInstaller);
Installers.Add(processInstaller);
```

The three essential properties that you have to define are the account to use to run the service, how and when it should be started – automatically or manually – and its name. Usually the LocalSystem account has enough privileges for a service, but you can supply a username and password with custom permissions. You also need to add a reference to System.Configuration.Install and add two 'Using' instructions to the start of the program:

```
using System.Configuration.Install;
using System.ComponentModel;
```

Now that we have all the code we need to create and install the service, we can try it out. First we need to build the project to create an executable in the Release folder – there's nothing to be gained or learned from running the application in Visual Studio as it has to be run as a service. Next we need to find the InstallUtil.exe program, open a command console and enter:

```
Path <directory that InstallUtil is
    stored in>
```

This will make InstallUtil accessible from any directory. Check that it has worked by entering:

```
InstallUtil
```

which should open a help screen explaining how to use the command. Finally, try the command:

```
InstallUtil /LogToConsole=true
SimpleServer\bin\Release\
SimpleServer.exe
```

You must supply the relative directory path to SimpleServer.exe.

You should see a lot of messages telling you that your service has been installed correctly. To check that it has worked, and to gain some familiarity with the Services tool, open the Control Panel, Administrative Tools, Services. Scroll down and you should find a new service called SimpleService in the list (see the screenshot, below left). You can use the tool to start and stop your service and modify it in other ways. If your service is not listed, then it hasn't been installed and you need to pay attention to the messages that InstallUtil has generated.

Once you have installed a service you can modify it simply by editing the code and building the project. This updates the EXE file stored in the release directory. To see the effect of your changes, simply restart the service. For example, we can bring up a message box to indicate that the service is starting. Change the OnStart event handler to read:

```
protected override void
    OnStart(string[] args)
{
```

```
    MessageBox.Show("SimpleService has
        started",
        "SimpleService",
        MessageBoxButtons.OK,
        MessageBoxIcon.Information,
        MessageBoxDefaultButton.Button1,
        MessageBoxOptions.
            ServiceNotification);
}
```

You also need to add a reference to System.Windows.Forms to the start of the program:

```
using System.Windows.Forms;
```

If you build the project and use the Services tool to restart the service, you should see the message box appear.

This special ServiceNotification message box is the only sort of user interface available to a basic service. You need to keep in mind that it will display even if no user is logged on, and only one such message box can be displayed at a time. Additional message boxes are queued and appear when the first one is dismissed.

You can give a service a full user interface using Windows Forms, but to do this you need to set AllowServiceToInteractWithDesktop. In this case you have to be careful to check that there is a user to interact with. There are also serious security problems with this type of service. If you find that you need a program with a user interface of this sort, you should seriously consider whether you really need a service.

USING TIMERS

In most cases a service creates an object or method that does the desired task when it starts and similarly destroys that object when it stops. The big problem in building any service is to get the object to do something when action is needed. There are two general approaches.

If there is an event that signals that there is work for the service to do, you can handle that event. For example, you can use the FileSystemWatcher to make the service respond to a change in the filing system. Alternatively you can implement a timer and wake the service up to do something at regular intervals. The timer approach is so common that it is worth looking at how it is done. Our final example shows how to work with an event.

First, let's create a timer object. We must add the following to the start of the program:

```
using System.Threading;
```

There are two timer classes, one in Windows.Forms and one in Threading. To make it clear which we want to use, we need to use a fully qualified class name when we declare a variable:

```
private System.Threading.Timer
    timer;
```

Now we can create the timer object and set it ticking when the service is started:

```
protected override void
    OnStart(string[] args)
{
    TimerCallback tick=new
        TimerCallback(SayHello);
    timer = new System.Threading.
        Timer(
```

ADDISON WESLEY The Old New Thing

★★★★

£27

Raymond Chen has worked with Windows since the days of MS-DOS and he knows a lot about its history and how it works. He has tried to capture his pearls of wisdom in *The Old New Thing*, which, as the title suggests, is a sort of history that explains why things are as they are. The problem is that the information is very bitty, ranging from pithy advice on designing a user interface to deep and technical mullings. Many of the topics are likely to be understood only by a C or C++ programmer who practised Windows programming back in the 'old days' from whence most of Chen's reminiscences stem.

How does this help with the "new thing"? If you're interested only in .NET development, you're



unlikely to gain much from Chen's advice. Those who understand how Windows uses messages, use a lot of API calls, know how dialog boxes work and have wrestled with old and new Windows memory management will find the book more rewarding. Both should enjoy the amusing anecdotes that provide a unique insight

into the Microsoft's workings and demonstrate what many of us have only guessed at – that Windows is a mess of compromises and unpleasant code implemented just to keep things working. A fun read but far from essential.

SUMMARY

- ✓ Insights into Microsoft
- ✗ Disorganised and chaotic

SUPPLIER www.amazon.co.uk
ISBN 0321440307
PAGES 560

```
tick, null, 0, 10000);  
}
```

We must remember to destroy the timer object when the service stops:

```
protected override void OnStop()  
{  
    timer.Dispose();  
}
```

Finally, we need to create the TimerCallback delegate. As a demonstration we'll make this delegate simply display the same message box we created earlier:

```
private void SayHello(object state)  
{  
    MessageBox.Show("Hello Service",  
        "SimpleService",  
        MessageBoxButtons.OK,  
        MessageBoxIcon.Information,  
        MessageBoxDefaultButton.Button1,  
        MessageBoxOptions.  
            ServiceNotification);  
}
```

If you start the service, a message box will pop up every 10 seconds until you stop the service. Stop the service to stop the messages.

A USEFUL SERVICE

The Event Log is the place to which a lot of important system components and applications send their status, warning and error messages. The problem is that most users either don't know about it or don't bother to look at it. There is an Event Viewer utility, found in Control Panel, Administrative Tools, that you can use to check that your machine is working properly, but it would be better if really important messages were brought more forcibly to your attention. We'll design a service called Eventer to do this. It will monitor the event log and email details of important new events as they occur.

To create Eventer all you have to do is create a new service called Eventer following the steps

given earlier. Start a new empty project called Eventer, add a code file called Eventer and add references to System, System.Configuration and System.ServiceProcess. Add the following to the start of the program:

```
using System;  
using System.ServiceProcess;  
using System.Configuration.Install;  
using System.ComponentModel;  
using System.Diagnostics;  
using System.Net.Mail;  
using System.Net;
```

The Eventer service starts in the usual way:

```
public class Eventer : ServiceBase  
{  
    public Eventer()  
    {  
        this.ServiceName = "Eventer";  
        this.CanStop = true;  
        this.CanPauseAndContinue =  
            false;  
        this.AutoLog = true;  
    }  
}
```

By setting the AutoLog property to true we make the service write start and stop events into the event log automatically.

The OnStart and OnStop methods are very simple because there is an EntryWrittenEvent, which we will use to activate the service – we won't need to use a timer. All we have to do to make the service respond to an event is add an event handler in the OnStart method and turn the events on:

```
protected override void  
OnStart(string[] args)  
{  
    this.EventLog.EntryWritten +=  
        new EntryWrittenEventHandler  
            (MyOnEntryWritten);  
    this.EventLog.EnableRaisingEvents  
        = true;  
}
```

For this project we can't use the form designer to connect an event to an event handler – it has to be done using code. This isn't difficult as all you need to do is add a suitable delegate to the event using the += operator. Of course, we still have to write the code in the MyOnEntryWritten event handler. In the OnStop method we need to remove the event handler, and this is just as easy using the -= operator on the event:

```
protected override void OnStop()  
{  
    this.EventLog.EnableRaisingEvents  
        = false;  
    this.EventLog.EntryWritten -=  
        new EntryWrittenEventHandler  
            (MyOnEntryWritten);  
}
```

This is all we need to get the service to start and stop, and the Main method has only to create an instance of the service to complete the code:

```
public static void Main()  
{  
    ServiceBase.Run(new Eventer());  
}
```

The installer for the service is similar to previous installers but this time the ServiceStartMode is set to Automatic, which starts the service as soon as the machine is started:

```
[RunInstaller(true)]  
public class EventerInstaller :  
    Installer  
{  
    private ServiceProcessInstaller  
        processInstaller;  
    private ServiceInstaller  
        serviceInstaller;  
  
    public EventerInstaller()  
    {  
        processInstaller = new  
            ServiceProcessInstaller();  
        serviceInstaller = new  
            ServiceInstaller();  
        processInstaller.Account =  
            ServiceAccount.LocalSystem;  
        serviceInstaller.StartType =  
            ServiceStartMode.Automatic;  
        serviceInstaller.ServiceName =  
            "Eventer";  
        Installers.  
            Add(serviceInstaller);  
        Installers.  
            Add(processInstaller);  
    }  
}
```

The only method we still need to write is the event handler that's called every time a new event is written to the Application event log. When you create a service object it has an EventLog property, which is connected to the local machine's Application event log. You can create other EventLog objects connected to other event logs – Security, System and so on if you want to scan for errors in specific categories – but, for the moment, working with the Application log provides an easy example. However, you can't change the EventLog property to work with any other event log; you have to create new EventLog objects.

The information about the entry that has just been written and so has caused the EntryWritten event is contained in the second parameter passed to the event handler. We could test the event type and set the service to email information about it only if it is serious enough – say, if it is at least a warning or an error. Again, for the sake of simplicity it is easier to email all events in the log initially, including low-priority information events. Our first task is to assemble the email that the service will send. Version 3 of the .NET framework makes this very easy with the introduction of a range of SMTP classes. First we create a MailMessage:

```
public void MyOnEntryWritten(
    Object source,
    EntryWrittenEventArgs e)
{
    MailMessage mail=new MailMessage
        ("from", "to");
```

replacing from and to with appropriate email addresses. You can also use the many properties MailMessage provides to set the subject and body, and even add attachments. In this case we simply need to set the subject and build up the text to use as the body of the email:

```
mail.Subject="Event Notification";
mail.Body=e.Entry.Source +
    + Environment.NewLine + e.Entry.
    Message +
    + Environment.NewLine +
    + e.Entry.TimeGenerated.
    ToLongDateString() +
    + " " + e.Entry.TimeGenerated.
    ToLongTimeString() +
    + Environment.NewLine + e.Entry.
    MachineName +
    + Environment.NewLine + e.Entry.
    EntryType;
```

The instructions that create the body of the email may look complicated but they simply build up the text using properties of e.Entry, such as the date, time and message of the log entry. You can include or leave out other details of the event to suit your requirements.

Now that the email is ready to send, we can use the SmtpClient to create a connection to an SMTP server. The documentation gives the impression that SmtpClient is to be used with an SMTP server provided by a local IIS virtual SMTP server; in fact, you can send email using any mail server on which you have a valid account. First we create an SmtpClient:

```
SmtpClient mailbox=new
    SmtpClient("smtp server");
```

You replace smtp server with either the IP address or URL of the mail server you plan to use. Some mail servers require you to log on before you can use them to send an email. If this is the case, you will need to set the Credentials property:

```
mailbox.Credentials =
    new NetworkCredential("user",
        "password");
```

where you replace user and password with a valid username and password. Finally we can send the mail message using the SMTP server:

```
mailbox.Send(mail);
}
```

This completes the event handler, and now all you need to do is install the service using InstallUtil. Once started, the service will send an email for each new event in the Application log.

Of course there are a few things we need to do to make the service robust, but this is a good first attempt. In particular we need to add some code to handle SMTP errors just in case the mail server is incorrectly configured or offline.

OTHER LOGS

Most of the important events relating to a PC's health are reported in the System log, so it's worth finding out how to extend Eventer's monitoring to the System and so to other logs. The only real difference is that to access another log we need to create an EventLog object:

```
EventLog SysLog;
```

OnStart now creates the EventLog object and sets its EntryWritten event handler to be the same as the one for the default Application log:

```
protected override void
    OnStart(string[] args)
{
    SysLog = new EventLog("System");
    SysLog.EntryWritten +=
        new EntryWrittenEventHandler
            (MyOnEntryWritten);
    this.EventLog.EntryWritten +=
        new EntryWrittenEventHandler
            (MyOnEntryWritten);
    this.EventLog.EnableRaisingEvents
        = true;
    SysLog.EnableRaisingEvents = true;
}
```

The OnStop method now has to remove the event handlers from both log objects:

```
protected override void OnStop()
{
    this.EventLog.EnableRaisingEvents
        = false;
    this.EventLog.EntryWritten -=
        new EntryWrittenEventHandler
            (MyOnEntryWritten);
    SysLog.EnableRaisingEvents =
        false;
    SysLog.EntryWritten -=
        new EntryWrittenEventHandler
            (MyOnEntryWritten);
}
```

Now if you start the service you will receive emails for each new entry in the Application and System logs. A more robust service would start a new thread to send each email, thereby making sure no events are missed, but the above code works well enough for most purposes.

CUSTOMISING A SERVICE

Rather than hard-coding the username, SMTP server and so on into a service, it would be better to provide some customisable command-line parameters. You may have noticed the args parameter in the OnStart method and wondered whether it can pass configuration information. It can, but it's not quite as simple as you'd think. To see it in action, add the following variables:

```
String SMTP;
String email;
String user;
String password;
```

Then add to the start of the OnStart method:

```
if (args.Length == 4)
{
    SMTP = args[0];
    email = args[1];
    user = args[2];
    password = args[3];
}
```

These variables can then be substituted for the hard-coded values in our service:

```
MailMessage mail =
    new MailMessage("Eventer@myPC",
        email);
```

and:

```
SmtpClient mailbox = new
    SmtpClient(SMTP);
mailbox.Credentials =
    new NetworkCredential(user,
        password);
```

Now you can specify where you want the emails sent as a command line within the Services tool. All you have to do is open the service's properties and enter the parameters, separated by spaces, in the 'Start Parameters' box.

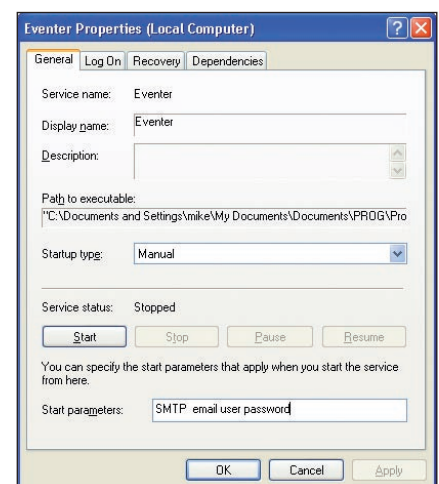
This is all very easy, but the surprise is that the service doesn't keep or use the parameters you enter the next time you start it. It uses them once and throws them away. If you want to supply fixed configuration parameters, you have to enter them in the Registry under the key:

```
HKEY_LOCAL_MACHINE\SYSTEM\
CurrentControlSet\Services\<service name>
```

You can then fetch these fixed parameters using:

```
Environment.GetCommandLineArgs();
```

This allows you to create a service capable of basic customisation. Again, if you require more regular interaction with the user, you probably don't want to implement your particular project as a service. **ES**



▲ Setting the service's start parameters